Programming up to Congruence

Vilhelm Sjöberg Stephanie Weirich

University of Pennsylvania, Philadelphia, PA, USA

{vilhelm,sweirich}@cis.upenn.edu

Abstract

This paper presents the design of ZOMBIE, a dependently-typed programming language that uses an adaptation of a congruence closure algorithm for proof and type inference. This algorithm allows the type checker to automatically use equality assumptions from the context when reasoning about equality. Most dependently-typed languages automatically use equalities that follow from β-reduction during type checking; however, such reasoning is incompatible with congruence closure. In contrast, ZOMBIE does not use automatic β-reduction because types may contain potentially diverging terms. Therefore ZOMBIE provides a unique opportunity to explore an alternative definition of equivalence in dependently-typed language design.

Our work includes the specification of the language via a bidirectional type system, which works "up-to-congruence," and an algorithm for elaborating expressions in this language to an explicitly typed core language. We prove that our elaboration algorithm is complete with respect to the source type system, and always produces well typed terms in the core language. This algorithm has been implemented in the ZOMBIE language, which includes general recursion, irrelevant arguments, heterogeneous equality and datatypes.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory

Keywords Dependent types; Congruence closure

1. Introduction

The ZOMBIE language [9] aims to provide a smooth path from ordinary functional programming in a language like Haskell to dependently typed programming in a language like Agda. However, one significant difference between Haskell and Agda is that in the latter, programmers must show that every function terminates. Such proofs often require delicate reasoning, especially when they must be done in conjunction with the function definition. In contrast, ZOMBIE includes arbitrary nontermination, relying on the type system to track whether an expression has been typechecked in the normalizing fragment of the language.

POPL '15, January 15–17, 2015, Mumbai, India.

ACM 978-1-4503-3300-9/15/01...$15.00.

http://dx.doi.org/10.1145/2676726.2676974

Prior work on ZOMBIE [9, 26] has focused on the metatheory of the core language—type safety for the entire language and consistency for the normalizing fragment—and provides a solid foundation. However, it is not feasible to write programs directly in the core language because the terms get cluttered with type annotations and type conversion proofs. This paper addresses the other half of the design: crafting a programmer-friendly *surface language*, which elaborates into the core.

The reason that elaboration is important in this context is that core ZOMBIE has a weak definition of equivalence. Most dependently-typed languages define terms to be equal when they are (at least) β-convertible. However, the presence of nontermination makes this definition awkward. To check whether two types are β-equivalent the type checker has to evaluate expressions inside them, which becomes problematic if expressions may diverge—what if the type checker gets stuck in an infinite loop? Existing languages fix an arbitrary global cut off for how many steps of evaluation the type-checker is willing to do (Cayenne [3]), or only reduce expressions that have passed a conservative termination test (Idris [8]). Core ZOMBIE, somewhat radically, omits automatic β-conversion completely. Instead, β-equality is available only through explicit conversion.

Because ZOMBIE does not include automatic β-conversion, it provides an opportunity to explore an alternative definition of equivalence in a surface language design.

Congruence closure, also known as the theory of equality with uninterpreted function symbols, is a basic operation in automatic theorem provers for first-order logic (particularly SMT solvers, such as Z3 [12]). Given some context Γ which contains assumptions of the form $a = b$, the congruence closure of Γ is the set of equations which are deducible by reflexivity, symmetry, transitivity, and changing subterms.

Although efficient algorithms for congruence closure are well-known [14, 19, 24] this reasoning principle has seen little use in *dependently-typed programming languages*. The problem is not lack of opportunity. Dependently-typed languages feature *propositional equality*, written $a = b$, which is a type that asserts the equality of the two expressions. Programs that use propositional equality build members of this type (using assumptions in the context, and various lemmas) and specify where and how they should be used. Congruence closure can assist with both of these tasks by automating the construction of these proofs and determining the "motive" for their elimination.

However, the adaption of this first-order technique to the higher-order logics of dependently-typed languages is not straightforward. The combination of congruence closure and full β-reduction makes the equality relation undecidable. As a result, most dependently-typed languages take the conservative approach of only incorporating congruence closure as a meta-operation, such as Coq's `congruence` tactic. While this tactic can assist with the creation of equality proofs, such proofs must still be explicitly eliminated. Proposals to use equations from the context automatically [1, 27, 28] have done so *in addition* to β-reduction, which makes it hard to characterize exactly which programs will typecheck, and also leaves open the question of how expressive congruence closure is in isolation.

In this work we define the ZOMBIE surface language to be fully "up to congruence", i.e. types which are equated by congruence closure can always be used interchangeably, and then show how the elaborator can implement this type system.

Designing a language around an elaborator—an unavoidably complicated piece of software—raises the risk of making the language hard to understand. Programmers could find it difficult to predict what core term a given surface term will elaborate to, or they may have to think about the details of the elaboration algorithm in order to understand whether a program will successfully elaborate at all.

We avoid these problems using two strategies. First, the syntax of the surface and the core language differ only by *erasable annotations* and the operational semantics ignores these annotations. Therefore the semantics of an expression is apparent just from looking at the source; the elaborator only adds annotations that can not change its behavior. Second, we define a *declarative specification* of the surface language, and prove that the elaborator is complete for the specification. As a result, the programmer does not have to think about the concrete elaboration algorithm.

We make the following contributions:

- We demonstrate how congruence closure is useful when programming, by comparing examples written in Agda, ZOMBIE, and ZOMBIE's explicitly-typed core language (Section 2).
- We define a dependently typed core language where the syntax contains erasable annotations (Section 3).
- We define a typed version of the congruence closure relation that is compatible with our core language, including features (erasure, injectivity, and generalized assumption) suitable for a dependent type system (Section 4).
- We specify the surface language using a bidirectional type system that uses this congruence closure relation as its *definition* of type equality (Section 5).
- We define an elaboration algorithm of the surface language to the core language (Section 6) based on a novel algorithm for typed congruence closure (Section 7). We prove that our elaboration algorithm is complete for the surface language and produces well-typed core language expressions. Our typed congruence closure algorithm both decides whether two terms are in the relation and also produces core language equality proofs.
- We have implemented these algorithms in ZOMBIE, extending the ideas of this paper to a language that includes datatypes and pattern matching, a richer logical fragment, and other features. Congruence closure works well in this setting; in particular, it significantly simplifies the typing rules for case-expressions (Section 8). Our implementation is available.[1]

For space reasons, the full specification of the type systems described in this paper and the details of the proofs are only included in the extended version [25].

[1] `https://code.google.com/p/trellys/`

2. Programming up to congruence

Consider this simple proof in Agda, which shows that zero is a right identity for addition.

```
npluszero : (n : Nat) → n + 0 ≡ n
npluszero zero    = refl
npluszero (suc m) = cong suc (npluszero m)
```

The proof follows by induction on natural numbers. In the base case, `refl` is a proof of `0 = 0`. In the next line, `cong` translates a proof of `m + 0 ≡ m` (from the recursive call) to a proof of `suc(m + 0) ≡ suc m`.

This proof relies on the fact that Agda's propositional equality relation (≡) is reflexive and a congruence relation. The former property holds by definition, but the latter must be explicitly shown. In other words, the proof relies on the following lemma:

```
cong : ∀ {A B} {m n : A}
        → (f : A → B) → m ≡ n → f m ≡ f n
cong f refl = refl
```

Now compare this proof to a similar result in ZOMBIE. The same reasoning is present: the proof follows via natural number induction, using the reduction behavior of addition in both cases.

```
npluszero : (n : Nat) → (n + 0 = n)
npluszero n =
  case n [eq] of
   Zero → (join : 0 + 0 = 0)
   Suc m →
     let _ = npluszero m in
       (join : (Suc m) + 0 = Suc (m + 0))
```

Because ZOMBIE does not provide automatic β-equivalence, reduction must be made explicit above. The term `join` explicitly introduces an equality based on reduction. However, in the successor case, the ZOMBIE type checker is able to infer exactly how the equalities should be put together.

For comparison, the corresponding ZOMBIE core language term includes a number of explicit type coercions:

```
npluszero : (n : Nat) → (n + 0 = n)
npluszero (n : Nat) =
  case n [eq] of
   Zero → join [⇝ 0 + 0 = 0]
            ▷ join [~eq + 0 = ~eq]
   Suc m →
     let ih = npluszero m in
      join [⇝ (Suc m) + 0 = Suc (m + 0)]
         ▷ join [(Suc m) + 0 = Suc ~ih]
         ▷ join [~eq + 0 = ~eq]
```

Above, an expression of the form $a \triangleright b$ converts the type of the expression a, using the equality proof b. Equality proofs may be formed in two ways, either via co-reduction (if a_1 and a_2 both reduce to some common term b, then `join`[⇝ $a_1 = a_2$] is a proof of their equality) or by congruence (if a is a proof of $b_1 = b_2$, then `join`[$\{\sim a/x\}A$] is a proof of $\{b_1/x\}A = \{b_2/x\}A$).

Both sorts of equality proofs are constructed in the example. In the base case, The proof `join[⇝ 0 + 0 = 0]` follows from reduction, and is converted to be a proof of `(n + 0) = n` by the congruence proof. Here, `eq` is a proof that `0 = n`, an assumption derived from pattern matching. Congruence reasoning constructs a

```
{-# NO_TERMINATION_CHECK #-}
unify : (t1 t2 : Term) → Unify t1 t2
unify leaf leaf = match empty refl
unify leaf (branch t2 t3) = nomatch
unify (branch t1 t2) leaf = nomatch
unify (branch t11 t12) (branch t21 t22)
      with unify t11 t21
...     | nomatch = nomatch
...     | match s p with unify (ap s t12) (ap s t22)
...                | nomatch = nomatch
...                | match s' q
  = match (compose s' s)
       (trans (apCompose (branch t11 t12))
         (trans (cong2 (λ t1 t2 →
                    branch (ap s' t1) t2) p q)
            (sym (apCompose (branch t21 t22)))))
unify t1 (var x) with (x is∈ t1)
...               | no q
  = match (singleton x t1)
       (trans (singleton-∉ t x t q)
          (varSingleton x t))
...               | yes p with t
...                       | var y
  = match empty (cong var (sym (invvar p)))
...                       | _
  = nomatch
unify (var x) t2 with unify t2 (var x)
...               | nomatch   = nomatch
...               | match s p = match s (sym p)
```

```
prog unify : (t1 t2 : Term) → Unify t1 t2
rec unify t1 = \ t2 . case t1, t2 of
    leaf, leaf → match empty _
    leaf, branch _ _ → nomatch
    branch _ _, leaf → nomatch
    branch t11 t12, branch t21 t22 →
      case (unify t11 t21) of
        nomatch → nomatch
        match s p → case (unify (ap s t12) (ap s t22)) of
            nomatch → nomatch
            match s' _ →
              unfold (ap s' (ap s t1)) in
              unfold (ap s' (ap s t2)) in
              let _ = apCompose s' s t1 in
              let _ = apCompose s' s t2 in
                 match (compose s' s) _
    _ , var x → case (isin x t1) of
        no q  →
           let _ = varSingleton x t1 in
           let _ = singletonNotIn t1 x t1 q in
             match (singleton x t1) _
        yes _ → case t1 of
           var y → let [_] = invvar x y p in
                     match empty _
           _     →
                       nomatch
    var x, _ →  case (unify t2 (var x)) of
        nomatch → nomatch
        match s p → match s _
```

Figure 1. First-order unification in Agda (left) and in ZOMBIE (right)

proof that that `((0 + 0) = 0) = ((n + 0) = n)`; the parts that differ on each side of the equality are marked by `~eq` in the congruence proof. The successor case uses congruence twice. The equality derived from reduction is first coerced by a congruence derived from the recursive call (`ih : m + 0 = m`), so that it has type `((Suc m) + 0 = Suc m)`. This equality is then coerced by a congruence derived from `eq : (Suc m = n)`, so that the result has type `(n + 0) = n`.

For a larger example, consider unification of first-order terms (Figure 1). For this example, the term language is the simplest possible, consisting only of binary trees constructed by `branch` and `leaf` and possibly containing unification variables, `var`, represented as natural numbers. We also use a type `Substitution` of substitutions, which are built by the functions `singleton` and `compose`, and applied to terms by `ap`.

Proving that `unify` terminates is difficult because the termination metric involves not just the structure of the terms but also the number of unassigned unification variables. (For example, see McBride [18]). To save development effort, a programmer may elect to prove only a partial correctness property: *if* the function terminates then the substitution it returns is a unifier.

In other words, if the `unify` function returns, it either says that the terms do not match, or produces a substitution `s` and a proof that `s` unifies them. We write the data structure in ZOMBIE as follows (the Agda version is similar):

```
data Unify (t1 : Term) (t2 : Term) : Type where
  nomatch
  match of (s : Substitution) (pf : ap s t1 = ap s t2)
```

Comparing the Agda and ZOMBIE implementations, we can see the effect of programming up-to-congruence instead of up-to-β. When the unifier returns `match`, it needs to supply a proof of equality. The Agda version explicitly constructs the proof using equational reasoning, which involves calling congruence lemmas `sym`, `trans`, `cong` and `cong`$_2$ from the standard library. The ZOMBIE version leaves such proof arguments as just an underscore, meaning that it can be inferred from the equations in the context. For that purpose, it introduces equalities to the context with `unfold` (for β-reductions, see Section 8.2) and with calls to relevant lemmas.

Figure 2 demonstrates how congruence closure makes ZOMBIE's version of dependently-typed pattern matching (i.e *smart case*) both simple and powerful. The figure compares (parts of) inductive proofs in ZOMBIE and Agda of an inversion lemma about the `snoc` operation, which appends an element to the end of a list. When both lists are nonempty, the proof argument can be used to derive that `x = y` (using the injectivity of `Cons`), and the recursive call shows that `xs' = ys'`. Congruence closure both puts these together in a proof of `Cons x xs' = Cons y ys'` and supplies the necessary proof for the recursive call.

In Agda, one is tempted to prove the property by pattern matching on the equality between the lists. This approach leads to a "quite fun" puzzle.[2] Here, the equivalence between `x` and `y` cannot be observed until `(snoc xs' z)` and `(snoc xs' z)` are named. The so-called "inspect on steroids" trick provides the equalities `(p : (snoc xs' z = s2)` and `q : (snoc ys' z) = s2)` that are necessary to constructing the fourth argument for the recursive call.

[2] Posed by Eric Mertens on `#agda`.

```
log snoc_inv : (xs ys : List A) → (z : A) → ((snoc xs z) = (snoc ys z)) → xs = ys
ind snoc_inv xs = \ ys z pf.  case xs [xeq], ys of
      Cons x xs' , Cons y ys' →
         let _ = (smartjoin : (snoc xs z) = Cons x (snoc xs' z)) in
         let _ = (smartjoin : (snoc ys z) = Cons y (snoc ys' z)) in
         let _ = snoc_inv xs' [ord xeq] ys' z _ in
           _
      ...

-- Agda pattern matching based solution
snoc-inv : ∀ xs ys z → (snoc xs z ≡ snoc ys z) → xs ≡ ys
snoc-inv (x :: xs')  (y :: ys') z pf with (snoc xs' z) | (snoc ys' z) | inspect (snoc xs') z | inspect (snoc ys') z
snoc-inv (.y :: xs')  (y :: ys') z refl    | .s2  | s2  | [ p ] | [ q ] with (snoc-inv xs' ys' z (trans p (sym q)))
snoc-inv (.y :: .ys') (y :: ys') z refl    | .s2  | s2  | [ p ] | [ q ]  | refl = refl
...

-- Alternative Agda solution based on congruence and injectivity
cons-inj1 : ∀ {x xs y ys} → ((x :: xs) ≡ y :: ys) → x ≡ y
cons-inj1 refl = refl

cons-inj2 : ∀ {x xs y ys} → x :: xs ≡ y :: ys → xs ≡ ys
cons-inj2 refl = refl

snoc-inv' : ∀ xs ys z → (snoc xs z ≡ snoc ys z) → xs ≡ ys
snoc-inv' (x :: xs') (y :: ys') z pf = cong₂ _::_ (cons-inj1 pf) (snoc-inv' xs' ys' z (cons-inj2 pf))
...
```

Figure 2. Pattern matching can be tricky in Agda

$$
\begin{array}{rcl}
x, y, f, g, h & \in & \text{expression variables} \\[6pt]
\text{expressions} & & \\
a, b, c, A, B & ::= & \mathsf{Type} \mid x \\
 & \mid & (x\!:\!A) \to B \mid \mathsf{rec}\ f_A\ x.a \mid a\ b \\
 & \mid & \bullet(x\!:\!A) \to B \mid \mathsf{rec}\ f_A\ \bullet_x.a \mid a\ \bullet_b \\
 & \mid & a = b \mid \mathsf{join}_\Sigma \mid a_{\triangleright b} \\[6pt]
\text{strategies} & & \\
\Sigma & ::= & \rightsquigarrow_{\mathsf{p}}\ i\ j : a = b \\
 & \mid & \{\sim v_1/x_1\} \ldots \{\sim v_j/x_j\} c : B \\
 & \mid & \mathsf{injdom}\ a \mid \mathsf{injrng}\ a\ b \mid \mathsf{injeq}\ i\ a \\[6pt]
\text{values} & & \\
v & ::= & \mathsf{Type} \mid x \\
 & \mid & (x\!:\!A) \to B \mid \mathsf{rec}\ f_A\ x.a \\
 & \mid & \bullet(x\!:\!A) \to B \mid \mathsf{rec}\ f_A\ \bullet_x.a \\
 & \mid & a = b \mid \mathsf{join}_\Sigma \mid v_{\triangleright b}
\end{array}
$$

Figure 3. Syntax

$\boxed{a \rightsquigarrow_{\mathsf{cbv}} b}$

$$\frac{}{(\mathsf{rec}\ f\ x.a)\ v \rightsquigarrow_{\mathsf{cbv}} \{v/x\}\{\mathsf{rec}\ f\ x.a/f\}\ a}\ \textsc{SCAppBeta}$$

$$\frac{}{(\mathsf{rec}\ f\ \bullet.a)\ \bullet \rightsquigarrow_{\mathsf{cbv}} \{\mathsf{rec}\ f\ \bullet.a/f\}\ a}\ \textsc{SCIAppBeta}$$

$$\frac{a \rightsquigarrow_{\mathsf{cbv}} a'}{a\ b \rightsquigarrow_{\mathsf{cbv}} a'\ b}\ \textsc{SCctx1}$$

$$\frac{a \rightsquigarrow_{\mathsf{cbv}} a'}{v\ a \rightsquigarrow_{\mathsf{cbv}} v\ a'}\ \textsc{SCctx2}$$

$$\frac{a \rightsquigarrow_{\mathsf{cbv}} a'}{a\ \bullet \rightsquigarrow_{\mathsf{cbv}} a'\ \bullet}\ \textsc{SCctx3}$$

Figure 4. Call-by-value operational semantics

Although this development is not long, it is not at all straightforward, requiring advanced knowledge of Agda idioms.

Alternatively, the reasoning used in the ZOMBIE example is also available in Agda, as in the definition of `snoc-inv'`. However, this version requires the use of helper functions to prove that cons is injective and congruent.

3. Annotated core language

We now turn to the theory of the system. We begin by describing the target of the elaborator: our annotated core language. This language is a small variant of the dependently-typed call-by-value language defined in prior work [26]. It corresponds to a portion of ZOMBIE's core language, but to keep the proofs tractable we omit ZOMBIE's recursive datatypes and replace its terminating sublanguage [9] with syntactic value restrictions.

The syntax is shown in Figure 3. Terms, types and the sort Type are collapsed into one syntactic category. We use the notation $\{a/x\}\ B$ to denote the capture-avoiding substitution of a for x in B. As is standard in dependently-typed languages, our notation for nondependent function types $A \to B$ is syntactic sugar for function types $(x\!:\!A) \to B$, where the variable x is not free in B.

Type *annotations*, such as A in rec $f_A\ x.a$, are optional and may be omitted from expressions. Annotations are subscripted in Figure 3. The meta-operator $|a|$ removes these annotations . Expressions that contain no typing annotations are called *erased*.

An expression that includes all annotations is called a *core* or *annotated* expression. The core typing judgement, written $\Gamma \vdash a : A$ and described below, requires that all annotations be present. In this case, the judgement is syntax-directed and trivially decidable. In contrast, type checking for erased terms is undecidable.

The only role of annotations is to ensure decidable type checking. They have no effect on the semantics. In fact, the operational semantics, written $a \rightsquigarrow_{\mathsf{cbv}} b$, is defined only for erased terms and extended to terms with annotations via erasure. This operational semantics is a small-step, call-by-value evaluation relation, shown in Figure 4.

Figure 5 shows the typing rules of the core language typing judgement $\Gamma \vdash a : A$. Additionally, the judgement $\vdash \Gamma$ (elided from the figure) states that each type in Γ is well-formed.

Recursive functions are defined using expressions rec $f\ x.a$, with the typing rule TREC. Such expressions are values, and applications step by the rule $(\mathsf{rec}\ f\ x.a)\ v \rightsquigarrow_{\mathsf{cbv}} \{v/x\}\{\mathsf{rec}\ f\ x.a/f\}\,a$. If the function makes no recursive calls we also use the syntactic sugar $\lambda x.a$. When a function has a dependent type (TDAPP) then its argument must be a value (this restriction is common for languages with nontermination [16, 30]).

Irrelevance In addition to the normal function type, the core language also include *computationally irrelevant* function types $\bullet(x : A) \to B$, which are inhabited by irrelevant functions rec $f_A\ \bullet_x.b$ and eliminated by irrelevant applications $a\ \bullet_b$. Many expressions in a dependently typed program are only used for type checking, but do not affect the runtime behavior of the program, and these can be marked irrelevant.

Our treatment of irrelevance follows ICC* [5]. Because the treatment of irrelevant functions closely mirrors that of normal functions, we omit the typing rules in this version of the paper. We include this feature in the formalism to show that, besides being generally useful, irrelevance works well with congruence closure. Given that we already handle erasable annotations, we can support full irrelevance for free.

Equality The typing rules at the bottom of Figure 5 deal with propositional equality, a primitive type. The formation rule TEQ states $a = b$ is a well-formed type whenever a and b are two well-typed expressions. There is no requirement that they have the *same* type (that is to say, our equality type is heterogeneous).

Propositional equality is eliminated by the rule TCAST: given a proof, v of an equation $A = B$ we can change the type of an expression from A to B. Since our equality is heterogeneous, we need to check that B is in fact a type. We require the proof to be a value in order to rule out divergence. A full-scale language could use a more ambitious termination analysis. (Indeed, our ZOMBIE implementation does so.) However, the congruence proofs generated by our elaborator are syntactic values, so for the purposes of this paper, the simple value restriction is enough. The proof term v in a type cast is an erasable annotation with no operational significance, so the typechecker considers equalities like $a = a_{\triangleright v}$ to be trivially true, and the elaborator is free to insert coercions using congruence closure proofs anywhere.

The rest of the figure shows introduction rules for equality. Equality proofs do not carry any information at runtime, so they all use the same term constructor join, but with different (erasable) annotations, Σ.

The rule TJOINP introduces equations which are justified by the operational semantics. ZOMBIE source programs must use TJOINP to explicitly indicate expressions that should be reduced. The rule states that join is a proof of $a_1 = a_2$ when the erasures of a_1 and a_2 reduce to a common expression b, using the parallel reduction relation. This common expression, b, is not required to be a value. Note that without normalization, we need a cutoff for how long to evaluate, so programmers must specify the number of steps i, j of reduction to allow (in ZOMBIE this defaults to 1000 if these numbers are elided).

The rule TJSUBST states that equality is a congruence. The simplest use of the rule is to change a single subexpression, using a proof v. The use of the proof is marked with a tilde in the Σ annotation; for example, if $\Gamma \vdash v : y = 0$ then we can prove the equality $\mathsf{join}_{\mathsf{Vec\ Nat}\ (\sim v) : \mathsf{Vec\ Nat}\ y = \mathsf{Vec\ Nat}\ 0}$. One can also eliminate several different equality proofs in one use of the rule. The syntax of subst includes a type annotation B, and the last premise of the TJSUBST rule checks that the ascribed type B matches what one gets after substituting the given equalities into the template c. This annotation adds flexibility because the check is only up-to erasure: if needed the programmer can give the left- and right-hand side of B different annotations to make both sides well-typed.

Finally, the rules TJINJEQ, TJINJDOM, and TJIINJRNG state that the equality type and arrow type constructors are injective. (The figure elides similar rules for irrelevant arrow types.) Making type constructors injective is unconventional for a dependent language. It is incompatible with e.g. Homotopy Type Theory, which proves $\mathsf{Nat} \to \mathsf{Void} = \mathsf{Bool} \to \mathsf{Void}$. However, in our language we need arrow injectivity to prove type preservation, because type casts are erased and do not block reduction [26]. For example, if a function coerced by type cast steps via β-reduction, we must use arrow injectivity to derive casts for the argument and result of the application.

We also add injectivity for the equality type constructor (TJINJEQ). This is not required for type safety, but it *is* justified by the metatheory, so it is safe to add. Injectivity is important for the surface language design, see Section 6.

The core language satisfies the usual properties for type systems. For the proofs in Section 6 we rely on the fact that it satisfies weakening, substitution (restricted to values), and regularity. It also satisfies preservation, progress, and decidable type checking. The proofs of these lemmas are in Sjöberg et al. [26].

4. Congruence closure

The driving idea behind our surface language is that the programmer should never have to explicitly write a type cast $a_{\triangleright v}$ if the proof v can be inferred by congruence closure. In this section we exactly specify which proofs can be inferred, by defining the typed congruence closure relation $\Gamma \vDash a = b$ shown in Figure 6.

Like the usual congruence closure relation for first-order terms, the rules in Figure 6, specify that this relation is reflexive, symmetric and transitive. It also includes rules for using assumptions in the context and congruence by changing subterms. However, we make a few changes:

First, we add typing premises (in TCCREFL and TCCERASURE) to make sure that the relation only equates well-typed and fully-annotated core language terms. In other words,

$\boxed{\Gamma \vdash a : A}$

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{Type} : \mathsf{Type}}\textsc{TType} \qquad \frac{x : A \in \Gamma \quad \vdash \Gamma}{\Gamma \vdash x : A}\textsc{TVar} \qquad \frac{\begin{array}{c}\Gamma \vdash A : \mathsf{Type}\\ \Gamma, x : A \vdash B : \mathsf{Type}\end{array}}{\Gamma \vdash (x\!:\!A) \to B : \mathsf{Type}}\textsc{TPi}$$

$$\frac{\begin{array}{c}\Gamma \vdash (x\!:\!A_1) \to A_2 : \mathsf{Type}\\ \Gamma, f : (x\!:\!A_1) \to A_2, x : A_1 \vdash a : A_2\end{array}}{\Gamma \vdash \mathsf{rec}\ f_{(x A_1) \to A_2}\ x.a : (x\!:\!A_1) \to A_2}\textsc{TRec} \qquad \frac{\begin{array}{c}\Gamma \vdash a : A \to B\\ \Gamma \vdash b : A\end{array}}{\Gamma \vdash a\ b : B}\textsc{TApp} \qquad \frac{\begin{array}{c}\Gamma \vdash a : (x\!:\!A) \to B\\ \Gamma \vdash v : A\end{array}}{\Gamma \vdash a\ v : \{v/x\}\, B}\textsc{TDApp}$$

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B}{\Gamma \vdash a = b : \mathsf{Type}}\textsc{TEq} \qquad \frac{\Gamma \vdash a : A \quad \Gamma \vdash v : A = B \quad \Gamma \vdash B : \mathsf{Type}}{\Gamma \vdash a_{\triangleright v} : B}\textsc{TCast}$$

$$\frac{|a_1| \rightsquigarrow_{\mathsf{p}}^{i} b \quad |a_2| \rightsquigarrow_{\mathsf{p}}^{j} b \quad \Gamma \vdash a_1 = a_2 : \mathsf{Type}}{\Gamma \vdash \mathsf{join}_{\rightsquigarrow_{\mathsf{p}} i\, j : a_1 = a_2} : a_1 = a_2}\textsc{TJoinP}$$

$$\frac{\begin{array}{c}\Gamma \vdash B : \mathsf{Type} \quad \forall k.\ \Gamma \vdash v_k : a_k = b_k\\ |B| = |(\{a_1/x_1\} \ldots \{a_j/x_j\}\, c = \{b_1/x_1\} \ldots \{b_j/x_j\}\, c)|\end{array}}{\Gamma \vdash \mathsf{join}_{\{\sim v_1/x_1\} \ldots \{\sim v_j/x_j\}\, c : B} : B}\textsc{TJSubst} \qquad \frac{\Gamma \vdash v : (A_1 = A_2) = (B_1 = B_2)}{\Gamma \vdash \mathsf{join}_{\mathsf{injeq}\ i\ v} : A_i = B_i}\textsc{TJInjEq}$$

$$\frac{\Gamma \vdash v : ((x\!:\!A_1) \to B_1) = ((x\!:\!A_2) \to B_2)}{\Gamma \vdash \mathsf{join}_{\mathsf{injdom}\ v} : A_1 = A_2}\textsc{TJInjDom} \qquad \frac{\Gamma \vdash v_1 : ((x\!:\!A) \to B_1) = ((x\!:\!A) \to B_2) \quad \Gamma \vdash v_2 : A}{\Gamma \vdash \mathsf{join}_{\mathsf{injrng}\ v_1\ v_2} : \{v_2/x\}\, B_1 = \{v_2/x\}\, B_2}\textsc{TJInjRng}$$

Figure 5. Typing rules for the annotated core language

$$\frac{\Gamma \vdash a : A}{\Gamma \vDash a = a}\textsc{TCCRefl} \qquad \frac{\Gamma \vDash a = b}{\Gamma \vDash b = a}\textsc{TCCSym} \qquad \frac{\Gamma \vDash a = b \quad \Gamma \vDash b = c}{\Gamma \vDash a = c}\textsc{TCCTrans}$$

$$\frac{|a| = |b| \quad \Gamma \vdash a : A \quad \Gamma \vdash b : B}{\Gamma \vDash a = b}\textsc{TCCErasure} \qquad \frac{x : A \in \Gamma \quad \Gamma \vDash A = (a = b)}{\Gamma \vDash a = b}\textsc{TCCAssumption}$$

$$\frac{\begin{array}{c}\Gamma \vdash A = B : \mathsf{Type} \quad \forall k.\ \Gamma \vDash a_k = b_k\\ |A = B| = |\{a_1/x_1\} \ldots \{a_j/x_j\}\, c = \{b_1/x_1\} \ldots \{b_j/x_j\}\, c|\end{array}}{\Gamma \vDash A = B}\textsc{TCCCongruence} \qquad \frac{\Gamma \vDash (a_1 = a_2) = (b_1 = b_2)}{\Gamma \vDash a_k = b_k}\textsc{TCCInjEq}$$

$$\frac{\Gamma \vDash ((x\!:\!A_1) \to B_1) = ((x\!:\!A_2) \to B_2)}{\Gamma \vDash A_1 = A_2}\textsc{TCCInjDom} \qquad \frac{\Gamma \vDash (A_1 \to B_1) = (A_2 \to B_2)}{\Gamma \vDash B_1 = B_2}\textsc{TCCInjRng}$$

Figure 6. Typed congruence closure relation

If $\Gamma \vDash a = b$, then $\Gamma \vdash a : A$ and $\Gamma \vdash b : B$.

Next, we adapt the congruence rule so that it corresponds to the TJSUBST rule of the core language. In particular, the rule TCCCONGRUENCE includes an explicit erasure step so that the two sides of the equality can differ in their erasable portions.

Furthermore, we extend the relation in several ways.[3] We automatically use computational irrelevance, in the rule TCCERASURE. This makes sure that the programmer can ignore all annotations when reasoning about programs. Also, we reason up to injectivity of datatype constructors (in rules TCCINJDOM, TCCINJRNG, and TCCINJEQ). As mentioned in Section 3 these rules are valid in the core language, and we will see in Section 6 that there is good reason to make the congruence closure algorithm use them automatically. Note that we restrict rule TCCINJRNG so that it applies only to nondependent function types; we explain this restriction in Section 6.

[3] Systems based around congruence closure often strengthen their automatic theorem prover in some way, e.g. Nieuwenhuis and Oliveras [20] add reasoning about natural number equations, and the Coq `congruence` tactic automatically uses injectivity of data constructors [10].

Finally, the rule TCCASSUMPTION is a bit stronger than the classic rule from first order logic. In the first-order logic setting, this rule is defined as just the closure over equations in the context:

$$\frac{x : a = b \in \Gamma}{\Gamma \vDash a = b}$$

However, in a dependently typed language, we can have equations between equations. In this setting, the classic rule does not respect CC-equivalence of contexts. For example, it would prove the first of the following two problem instances, but not the second.

$$x\!:\!\mathsf{Nat}, y\!:\!\mathsf{Nat}, a\!:\!\mathsf{Type}, h_1\!:\!(x = y) = a, h_2\!:\!x = y \vDash x = y$$

$$x\!:\!\mathsf{Nat}, y\!:\!\mathsf{Nat}, a\!:\!\mathsf{Type}, h_1\!:\!(x = y) = a, h_2\!:\!a \qquad \vDash x = y$$

Therefore we replace the rule with the stronger version shown in the figure.

We were led to these strengthened rules by theoretical considerations when trying to show that our elaboration algorithm was complete with respect to the declarative specification (see Section 6). Once we implemented the current set of rules, we found that they were useful in practice as well as in theory, because they improved the elaboration of some examples in our test suite. The stronger assumption rule is useful in situations where type-level computation produces equality types, for example when using custom induction principles.

5. Surface language

Next, we give a precise specification of the surface language, which shows how type inference can use congruence closure to infer casts of the form $a_{\triangleright v}$. Note that this process involves determining both the location of such casts and the proof of equality v.

Figure 7 defines a *bidirectional type system* for a partially annotated language. This type system is defined by two (mutually defined) judgements: *type synthesis*, written $\Gamma \vdash a \Rightarrow A$, and *type checking*, written $\Gamma \vdash a \Leftarrow A$. Here Γ and a are always inputs, but A is an output of the synthesizing judgement and an input of the checking judgement.

Most rules of this type system are standard for bidirectional systems [22], including the rules for inferring the types of variables (IVAR), the well-formedness of types (IEQ, ITYPE, and IPI), non-dependent application (IAPP), and the mode switching rules CINF and IANNOT. Any term that has enough annotations to synthesize a type A also checks against that type (CINF). Conversely, some terms (e.g. functions) require a known type to check against, and so if the surrounding context does not specify one, the programmer must add a type annotation (IANNOT).

The rules ICAST and CCAST in Figure 7 specify that checking and inference work "up-to-congruence." At any point in the typing derivation, the system can replace the inferred or checked type with something congruent. The notation $\Gamma \vDash^{\exists} A = B$ lifts the congruence closure judgement from Section 4 to the partially annotated surface language. These two rules contain kinding premises to maintain well-formedness of types. The invariant maintained by the type system is that (in a well-formed context Γ) any synthesized type is guaranteed to be well-kinded, while it is the caller's responsibility to ensure that any time the checking judgement is used the input type is well-kinded.

The rule for checking functions (CREC) is almost identical to the corresponding rule in the core language, with just two changes. First, the programmer can omit the types A_1, and A_2, because in a bidirectional system they can be deduced from the type the expression is checked against. Second, the new premise injrng slightly restricts the use of this rule. The difficulty is that the congruence closure algorithm does not implement the full TJINJRNG rule of the core language, but injective reasoning is needed by the type checker. Therefore, we rule out function types that do not support injectivity for their ranges in certain (pathological) typing contexts. This premise also appears in the rule for dependent application (IDAPP). We return to this issue in Section 6.

Equations that are provable via congruence closure are available via the checking rule, CREFL. In this case the proof term is just join, written as an underscore in the concrete syntax. Because this is a checking rule, the equation to be proved does not have to be written down directly if it can be inferred from the context.

The rule IJOINP proves equations using the operational semantics. We saw this rule used in the `npluszero` example, written `join : 0 + 0 = 0` in the concrete syntax. Note that the programmer must explicitly write down the terms that should be reduced. The rule IJOINP is a synthesizing rather than checking rule in order to ensure that the typing rules are effectively implementable. Although the type system works "up to congruence" the operational semantics do not. So the expression itself needs to contain enough information to tell the typechecker which member of the equivalence class should be reduced—it cannot get this information from the checking context. (In practice, having to explicitly write this annotation can be annoying. The ZOMBIE implementation includes a feature `smartjoin` which can help—see Section 8.2).

It is also interesting to note the rules that do *not* appear in Figure 7. For example, there is no rule or surface syntax corresponding to TCAST, because this feature can be written as a user-level function. Similarly, the rather involved machinery for rewriting subterms and and erased terms (rule TJSUBST) can be entirely omitted, since it is subsumed by the congruence closure relation. The programmer only needs to introduce the equations into the context and they will be used automatically.

Finally we note that the surface language does not satisfy some of the usual properties of type systems. In particular, it lacks a general weakening lemma because the injrng relation cannot be weakened. Similarly, it does not satisfy a substitution lemma because that property fails for the congruence closure relation. (We might expect that $\Gamma, x : C \vDash a = b$ and $\Gamma \vdash v : C$ would imply $\Gamma \vDash \{v/x\}\, a = \{v/x\}\, b$. But this fails if C is an equation and the proof v makes use of the operational semantics.) And it does not satisfy a strengthening lemma, because even variables that do not occur in a term may be implicitly used as assumptions of congruence proofs.

The situations where weakening and substitution fail are rare (we have never encountered one when writing example programs in ZOMBIE) and there are straightforward workarounds for programmers. Furthermore, these properties do hold for fully annotated expressions, so there are no restrictions on the output of elaboration. However, the typing rules for the declarative system must be formulated to avoid these issues, which requires some extra premises. The rule IVAR requires $\Gamma \vdash A \Leftarrow \mathsf{Type}$ (proving this from $\vdash \Gamma \Leftarrow$ would need weakening); IAPP requires $\Gamma \vdash B \Leftarrow \mathsf{Type}$ (proving this from $\Gamma \vdash A \to B : \mathsf{Type}$ would need strengthening); and CREC requires $\Gamma, f : (x : A_1) \to A_2 \vdash (x : A_1) \to A_2 \Leftarrow \mathsf{Type}$ (proving this from $\Gamma \vdash (x : A_1) \to A_2 \Leftarrow \mathsf{Type}$ would need weakening).

6. Elaboration

We implement the declarative system using an elaborating typechecker, which translates a surface language expression (if it is well-formed according to the bidirectional rules) to an expression in the core language.

We formalize the algorithm that the elaborator uses as two inductively defined judgements, written $\Gamma' \mapsto a \Rightarrow a' : A'$ (Γ' and a are inputs) and $\Gamma' \mapsto a \Leftarrow A' \rightsquigarrow a'$ (Γ', a, and A' are inputs). The variables with primes (Γ', a' and A') are fully annotated expressions in the core language, while a is the surface language term being elaborated. The elaborator deals with each top-level definition in the program separately, and the context Γ' is an input containing the types of the previously elaborated definitions.

The job of the elaborator is to insert enough annotations in the term to create a well-typed core expression. It should not otherwise change the term. Stated more formally,

***Theorem* 1** (Elaboration soundness)**.**

1. If $\Gamma \mapsto a \Rightarrow a' : A$, then $\Gamma \vdash a' : A$ and $|a| = |a'|$.
2. If $\Gamma \vdash A : \mathsf{Type}$ and $\Gamma \mapsto a \Leftarrow A \rightsquigarrow a'$, then $\Gamma \vdash a' : A$ and $|a| = |a'|$.

$$\boxed{\Gamma \vdash a \Rightarrow A}$$

$$\frac{\Gamma \vdash a \Rightarrow A \quad \Gamma \vDash^{\exists} A = B \quad \Gamma \vdash B \Leftarrow \mathsf{Type}}{\Gamma \vdash a \Rightarrow B}\ \textsc{ICast}$$

$$\frac{\Gamma \vdash A \Leftarrow \mathsf{Type} \quad \Gamma \vdash a \Leftarrow A}{\Gamma \vdash a_A \Rightarrow A}\ \textsc{IAnnot}$$

$$\frac{\Gamma \vdash a \Rightarrow A \quad \Gamma \vdash b \Rightarrow B}{\Gamma \vdash a = b \Rightarrow \mathsf{Type}}\ \textsc{IEq}$$

$$\frac{\begin{array}{c}\Gamma \vdash a_1 = a_2 \Leftarrow \mathsf{Type} \\ |a_1| \leadsto_{\mathsf{p}}^{i} b \quad |a_2| \leadsto_{\mathsf{p}}^{j} b\end{array}}{\Gamma \vdash \mathsf{join}_{\leadsto_{\mathsf{p}} i\, j : a_1 = a_2} \Rightarrow a_1 = a_2}\ \textsc{IJoinP}$$

$$\frac{\vdash \Gamma \Leftarrow}{\Gamma \vdash \mathsf{Type} \Rightarrow \mathsf{Type}}\ \textsc{IType}$$

$$\frac{\vdash \Gamma \Leftarrow \quad x : A \in \Gamma \quad \Gamma \vdash A \Leftarrow \mathsf{Type}}{\Gamma \vdash x \Rightarrow A}\ \textsc{IVar}$$

$$\frac{\Gamma \vdash A \Leftarrow \mathsf{Type} \quad \Gamma, x : A \vdash B \Leftarrow \mathsf{Type}}{\Gamma \vdash (x : A) \to B \Rightarrow \mathsf{Type}}\ \textsc{IPi}$$

$$\frac{\begin{array}{l}\Gamma \vdash a \Rightarrow A \to B \quad \Gamma \vdash b \Leftarrow A \\ \Gamma \vdash B \Leftarrow \mathsf{Type}\end{array}}{\Gamma \vdash a\ b \Rightarrow B}\ \textsc{IApp}$$

$$\frac{\begin{array}{l}\Gamma \vdash a \Rightarrow (x : A) \to B \quad \Gamma \vdash v \Leftarrow A \\ \Gamma \vDash^{\exists} \mathsf{injrng}\,(x : A) \to B \text{ for } v \\ \Gamma \vdash \{v_A/x\}\, B \Leftarrow \mathsf{Type}\end{array}}{\Gamma \vdash a\ v \Rightarrow \{v_A/x\}\, B}\ \textsc{IDApp}$$

$$\boxed{\Gamma \vdash a \Leftarrow A}$$

$$\frac{\Gamma \vdash a \Leftarrow A \quad \Gamma \vDash^{\exists} A = B \quad \Gamma \vdash B \Leftarrow \mathsf{Type}}{\Gamma \vdash a \Leftarrow B}\ \textsc{CCast}$$

$$\frac{\Gamma \vdash a \Rightarrow A}{\Gamma \vdash a \Leftarrow A}\ \textsc{CInf}$$

$$\frac{\Gamma \vDash^{\exists} a = b}{\Gamma \vdash \mathsf{join} \Leftarrow a = b}\ \textsc{CRefl}$$

$$\frac{\begin{array}{l}\Gamma, f : (x : A_1) \to A_2, x : A_1 \vdash a \Leftarrow A_2 \\ \Gamma, f : (x : A_1) \to A_2, x : A_1 \vDash^{\exists} \mathsf{injrng}\,(x : A_1) \to A_2 \text{ for } x \\ \Gamma, f : (x : A_1) \to A_2 \vdash (x : A_1) \to A_2 \Leftarrow \mathsf{Type}\end{array}}{\Gamma \vdash \mathsf{rec}\ f\ x.a \Leftarrow (x : A_1) \to A_2}\ \textsc{CRec}$$

$$\boxed{\Gamma \vDash^{\exists} a = b}$$

$$\frac{\Gamma' \vDash a' = b' \quad |a| = |a'| \quad |b| = |b'| \quad |\Gamma| = |\Gamma'|}{\Gamma \vDash^{\exists} a = b}\ \textsc{EEq}$$

$$\boxed{\Gamma \vDash^{\exists} \mathsf{injrng}\ A \text{ for } v}$$

$$\frac{\begin{array}{c}\forall A' B'.\ \Gamma \vDash^{\exists} ((x : A) \to B) = ((x : A') \to B') \\ \text{implies } \Gamma \vDash^{\exists} \{v/x\}\, B = \{v/x\}\, B'\end{array}}{\Gamma \vDash^{\exists} \mathsf{injrng}\,(x : A) \to B \text{ for } v}\ \textsc{EIRPi}$$

Figure 7. Bidirectional typing rules for surface language

$$\boxed{\Gamma \vDash \mathsf{injrng}\ A \text{ for } v}$$

$$\frac{\begin{array}{l}\Gamma \vdash v : A \quad \Gamma \vdash (x : A) \to B : \mathsf{Type} \\ \forall A'\ B'.((\Gamma \vDash ((x : A) \to B) = ((x : A') \to B')) \text{ implies } (\Gamma \vDash \{v/x\}\, B = \{v_{\triangleright v_0}/x\}\, B' \text{ where } \Gamma \vdash v_0 : A = A'))\end{array}}{\Gamma \vDash \mathsf{injrng}\,(x : A) \to B \text{ for } v}\ \textsc{IRPi}$$

Figure 8. Core language injectivity restriction

Furthermore, the elaborator should accept those terms specified by the declarative system. If the type system of Section 5 accepts a program, then the elaborator succeeds (and produces an equivalent type in inference mode).

***Theorem* 2** (Elaboration completeness)**.**

1. If $\Gamma \vdash a \Rightarrow A$ and $\Vdash \Gamma \leadsto \Gamma'$ and $\Gamma' \Vdash A \Leftarrow \mathsf{Type} \leadsto A'$, then $\Gamma' \Vdash a \Rightarrow a' : A''$ and $\Gamma' \vDash A' = A''$

2. If $\Gamma \vdash a \Leftarrow A$ and and $\Vdash \Gamma \leadsto \Gamma'$ and $\Gamma' \Vdash A \Leftarrow \mathsf{Type} \leadsto A'$, then $\Gamma' \Vdash a \Leftarrow A' \leadsto a'$.

Designing the elaboration rules follows the standard pattern of turning a declarative specification into an algorithm: remove all rules that are not syntax directed (in this case ICast and CCast), and generalize the premises of the remaining rules to create a syntax-directed system that accepts the same terms. At the same time, the uses of congruence closure relation $\Gamma \vDash a = b$, must be replaced by appropriate calls to the congruence closure algorithm. We specify this algorithm using the following (partial) functions:

- $\Gamma \Vdash A \stackrel{?}{=} B \leadsto v$, which checks A and B for equality and produces core-language proof v.
- $\Gamma \Vdash A =^{?} (x : B_1) \to B_2 \leadsto v$, which checks whether A is equal to some function type and produces that type and proof v.
- $\Gamma \Vdash A =^{?} (B_1 = B_2) \leadsto v$, which is similar to above, except for equality types.

For example, consider the rule for elaborating function applications:

$$\frac{\begin{array}{l}\Gamma \Vdash a \Rightarrow a' : A_1 \quad \Gamma \Vdash A_1 =^{?} (x : A) \to B \leadsto v_1 \\ \Gamma \Vdash v \Leftarrow A \leadsto v' \quad \Gamma \vDash \mathsf{injrng}\,(x : A) \to B \text{ for } v'\end{array}}{\Gamma \Vdash a\ v \Rightarrow a'_{\triangleright v_1}\ v' : \{v'/x\}\, B}\ \textsc{EIDApp}$$

In the corresponding declarative rule (IDAPP) the applied term a must have an arrow type, but this can be arranged by implicitly using ICAST to adjust a's type. Therefore, in the algorithmic system, the corresponding condition is that the type of a should be equal to an arrow type $(x : A) \to B$ modulo the congruence closure. Operationally, the typechecker will infer some type A_1 for a, then run the congruence closure algorithm to construct the set of all expressions that are equal to A_1, and check if the set contains some expression which is an arrow type. The elaborated core term uses the produced proof of $A_1 = (x : A) \to B$ in a cast to change the type of a.

At this point there is a potential problem: what if A_1 is equal to more than one arrow type? For example, if $A_1 = (x : A) \to B = (x : A') \to B$, then the elaborator has to choose whether to check b against A or A'. A priori it is quite possible that only one of them will work; for example the context Γ may contain an inconsistent equation like $\mathsf{Nat} \to \mathsf{Nat} = \mathsf{Bool} \to \mathsf{Nat}$. We do not wish to introduce a backtracking search here, because that could make type checking too slow.

This kind of mismatch in the domain type can be handled by extending the congruence closure algorithm. Note that things are fine if $\Gamma \vDash A = A'$, since then it does not matter if A or A' is chosen. So the issue only arises if $\Gamma \vDash (x : A) \to B = (x : A') \to B$ and not $\Gamma \vDash A = A'$. Fortunately, type constructors are injective in the core language (Section 3). Including injectivity as part of the congruence closure judgement (by the rule TCCINJDOM) ensures that it does not matter which arrow type is picked.

We also have to worry about a mismatch in the codomain type, i.e. the case when $\Gamma \vDash A_1 = (x : A) \to B$ and $\Gamma \vDash A_1 = (x : A') \to B'$ for two different types. At first glance it seems as if we could use the same solution. After all, the core language includes a rule for injectivity of the range of function types (rule TJINJRNG). There is an important difference between this rule and TJINJDOM, however, which is the handling of the bound variable x in the codomain B: the rule says that this can be closed by substituting any value for it. As a result, we cannot match this rule in the congruence closure relation, because the algorithm would have to guess that value. As far as writing an elaborator goes, maybe this is fine—after all, we only want to apply the axiom to the particular value v from the function application $a\ v$. However, there does not seem to be any natural way to write a declarative specification explaining what values v should be candidates.

Instead, we restrict the declarative language to forbid this problematic case. That is, the programmer is not allowed to write a function application unless all possible return types for the function are equal. Note that in cases when an application is forbidden by this check, the programmer can avoid the problem by proving the required equation manually and ensuring that it is available in the context.

In the fully-annotated core language we express this restriction with the rule IRPI (in Figure 8), and then lift this operation to partially annotated terms by rule EIRPI (Figure 7). Operationally, the typechecker will search for all arrow types equal to A_1 and check that the the codomains with v substituted are equal in the congruence closure. This takes advantage of the fact that equivalence classes under congruence closure can be efficiently represented—although the rule as written appears to quantify over potentially infinitely many function types, the algorithm in Section 7 will represent these as a finite union-find structure which can be effectively enumerated. In the core language rule we need to insert a type coercion from A to A' to make the right-hand side well typed. By the rule TCCINJDOM that equality is always provable, so the typechecker will use the proof term v_0 that the congruence closure algorithm produced.

On the checking side, the mode-change rule ECINF now needs to prove that the synthesized and checked types are equal.

$$\frac{\Gamma \vdash a \Rightarrow a' : A \quad \Gamma \vdash A \stackrel{?}{=} B \rightsquigarrow v_1}{\Gamma \vdash a \Leftarrow B \rightsquigarrow {a'}_{\triangleright v_1}}\ \text{ECINF}$$

This rule corresponds to a direct call to the congruence closure algorithm, producing a proof term v_1. Note that the inputs are fully elaborated terms—in moving from the declarative to the algorithmic type system, we replaced the undecidable condition $\Gamma \vDash^{\exists} A = B$ with a decidable one.

Finally, the rule ECREFL elaborates checkable equality proofs (written as underscores in the concrete ZOMBIE syntax).

$$\frac{\Gamma \vdash A =^{?} (a = b) \rightsquigarrow v_1 \quad \Gamma \vdash a \stackrel{?}{=} b \rightsquigarrow v}{\Gamma \vdash \mathsf{join} \Leftarrow A \rightsquigarrow v_{\triangleright \mathsf{symm}\ v_1}}\ \text{ECREFL}$$

As in the rule for application, the typechecker does a search through the equivalence class of the ascribed type A to see if it contains any equations. If there is more than one equation it does not matter which one gets picked, because the congruence relation includes injectivity of the equality type constructor (TCCINJEQ). In the elaborated term we need to prove $(a = b) = A$ given $A = (a = b)$. This can be done using TJOINP (for reflexivity) and TJSUBST, and we abbreviate that proof term $\mathsf{symm}\ v_1$.

7. Implementing congruence closure

Algorithms for congruence closure in the first-order setting are well studied, and our work builds on them. However, in our type system the relation $\Gamma \vDash a = b$ does more work than "classic" congruence closure: we must also handle erasure, terms with bound variables, (dependently) typed terms, the injectivity rules, the "assumption up to congruence" rule, and we must generate proof terms in the core language.

Our implementation proves an equation $a = b$ in three steps. First, we erase all annotations from the input terms and explicitly mark places where the congruence rule can be applied, using an operation called *labelling*. Then we use an adapted version of the congruence closure algorithm by Nieuwenhuis and Oliveras [20]. Our version of their algorithm has been extended to also handle injectivity and "assumption up to congruence", but it ignores all the checks that the terms involved are well-typed. Finally, we take the untyped proof of equality, and process it into a proof that a and b are also related by the typed relation. The implementation is factored in this way because the congruence rule does not necessarily preserve well-typedness, so the invariants of the algorithm are easier to maintain if we do not have to track well-typedness at the same time.

7.1 Labelling terms

In $\Gamma \vDash a = b$, the rule TCCCONGRUENCE is stated in terms of substitution. But existing algorithms expect congruence to be applied only to syntactic function applications: from $a = b$ conclude $f\ a = f\ b$. To bridge this gap, we preprocess equations into (erased) *labelled expressions*. A label F is an erased language expression with some designated holes (written –) in it, and a labelled expression is a label applied to zero or more labelled expressions, i.e. a term in the following grammar.

$$a ::= F\,\overline{a_i}$$

$$\frac{}{\Gamma \vdash^{\mathsf{L}} a = a}\textsc{LCCRefl} \qquad \frac{\Gamma \vdash^{\mathsf{L}} a = b}{\Gamma \vdash^{\mathsf{L}} b = a}\textsc{LCCSym} \qquad \frac{\Gamma \vdash^{\mathsf{L}} a_1 = b \quad \Gamma \vdash^{\mathsf{L}} b = a_2}{\Gamma \vdash^{\mathsf{L}} a_1 = a_2}\textsc{LCCTrans}$$

$$\frac{x : A \in \Gamma \quad \Gamma \vdash^{\mathsf{L}} A = ((- = -)\ a\ b)}{\Gamma \vdash^{\mathsf{L}} a = b}\textsc{LCCAssum} \qquad \frac{\forall k.\ \Gamma \vdash^{\mathsf{L}} a_k = b_k}{\Gamma \vdash^{\mathsf{L}} F\,\overline{a_i} = F\,\overline{b_i}}\textsc{LCCCong} \qquad \frac{\Gamma \vdash^{\mathsf{L}} F\ a = F\ b \quad F\ \mathsf{injective}}{\Gamma \vdash^{\mathsf{L}} a = b}\textsc{LCCInj}$$

Figure 9. Untyped congruence closure on labelled terms

Typically a label will represent just a single node of the abstract syntax tree. For example, a wanted equation $f\ x = f\ y$ will be processed into $(-\ -)\ f\ x = (-\ -)\ f\ y$, where the label $(-\ -)$ means this is an application. However, for syntactic forms involving bound variables, it can be necessary to be more coarse-grained. For example, given $a = b$ our implementation can prove $\mathsf{rec}\ f\ x.a + x = \mathsf{rec}\ f\ x.b + x$, which involves using $\mathsf{rec}\ f\ x.- + x$ as a label. In general, to process an expression a into a labelled term, the implementation will select the largest subexpressions that do not involve any bound variables.

Applying the labelling step simplifies the congruence closure problems in several ways. We show the simpler problem by defining the relation $\Gamma \vdash^{\mathsf{L}} a = b$ defined in Figure 9. Compared to Figure 6 we no longer need a rule for erasure, congruence is only used on syntactic label applications, all the different injectivity rules are handled generically, and we do not ensure that the terms are well-typed. In the appendix we formally define the label operation, and prove that it is complete in the following sense.

***Lemma* 3.** If $\Gamma \vDash a = b$ then $\mathsf{label}\,\Gamma \vdash^{\mathsf{L}} \mathsf{label}\ a = \mathsf{label}\ b$.

7.2 Untyped congruence closure

Next, we use an algorithm based on Nieuwenhuis and Oliveras [20] to decide the $\Gamma \vdash^{\mathsf{L}} a = b$ relation. The algorithm first "flattens" the problem by allocating *constants* c_i (i.e. fresh names) for every subexpression in the input. After this transformation every input equation has either the form $c_1 = c_2$ or $c = F(c_1, c_2)$, that is, it is either an equation between two atomic constants or between a constant and a label F applied to constants. Then follows the main loop of the algorithm, which is centered around three data-structures: a queue of input equations, a union-find structure and a lookup table. In each step of the loop, we take off an equation from the queue and update the state accordingly. When all the equations have been processed the union-find structure represents the congruence closure.

The union-find structure tracks which constants are known to be equal to each other. When the algorithm sees an input equation $c_1 = c_2$ it merges the corresponding union-find classes. This deals with the reflexivity, symmetry and transitivity rules. The lookup table is used to handle the congruence rule. It maps applications $F(c_1, c_2)$ to some canonical representative c. If the algorithm sees an input equation $c = F(c_1, c_2)$, then c is recorded as the representative. If the table already had an entry c', then we deduce a new equation $c = c'$ which is added to the queue.

In order to adapt this algorithm to our setting, we make three changes. First, we adapt the lookup tables to include the *richer labels* corresponding to the many syntactic categories of our core language. (Nieuwenhuis and Oliveras only use a single label meaning "application of a unary function.")

Second, we deal with *injectivity rules* in a way similar to the implementation of Coq's `congruence` tactic [10]. Certain labels are considered injective, and in each union-find class we identify the set of terms that start with an injective label. If we see an input equation $c = F(c_1, c_2)$ and F is injective we record this in the class of c. Whenever we merge two classes, we check for terms headed by the same F; e.g. if we merge a class containing $F(c_1, c_2)$ with a class containing $F(c_1', c_2')$, we deduce new equations $c_1 = c_1'$ and $c_2 = c_2'$ and add those to the queue.

Third, our implementation of the *extended assumption rule* works much like injectivity. With each union-find class we record two new pieces of information: whether any of the constants in the class (which represent types of our language) are known to be inhabited by a variable, and whether any of the constants in the class represents an equality type. Whenever we merge two classes we check for new equations to be added to the queue.

The extended version of the paper contains a precise description of our algorithm, and also gives a formal proof of its correctness:

***Lemma* 4.** The algorithm described above is a decision procedure for the relation $\Gamma \vdash^{\mathsf{L}} a = b$.

7.3 Typing restrictions and generating core language proofs

Along the pointers in the union-find structure, we also keep track of the evidence that showed that two expressions are equal. The syntax of the evidence terms is given by the following grammar. An evidence term p is either an assumption x (with a proof p that x's type is an equation), reflexivity, symmetry, transitivity, injectivity, or an application of congruence annotated with a label A.

$$p, q \quad ::= \quad x_{\triangleright p} \mid \mathsf{refl} \mid p^{-1} \mid p; q \mid \mathsf{inj}_i\ p \mid \mathsf{cong}_A\ p_1 \mathrel{..} p_i$$

Next we need to turn the evidence terms p into proof terms in the core calculus. This is nontrivial, because the Nieuwenhuis-Oliveras algorithm does not track types. Not every equation which is derivable by untyped congruence closure is derivable in the typed theory; for example, if $f : \mathsf{Bool} \to \mathsf{Bool}$, then from the equation $(a : \mathsf{Nat}) = (b : \mathsf{Nat})$ we can not conclude $f\ a = f\ b$, because $f\ a$ is not a well-typed term. Worse still, even if the conclusion is well-typed, not every untyped *proof* is valid in the typed theory, because it may involve ill-typed intermediate terms. For example, let $\mathsf{Id} : (A : \mathsf{Type}) \to A \to A$ be the polymorphic identity function, and suppose we have some terms $a : A$, $b : B$, and know the equations $x : A = B$ and $y : a = b$. Then

$$(\mathsf{cong}_{\mathsf{Id}}\ x\ \mathsf{refl}); (\mathsf{cong}_{\mathsf{Id}}\ \mathsf{refl}\ y)$$

is a valid untyped proof of $\mathsf{Id}\ A\ a = \mathsf{Id}\ B\ b$. But it is not a correct typed proof because it involves the ill-typed term $\mathsf{Id}\ B\ a$:

$$\frac{\dfrac{x : A = B \qquad a = a}{\mathsf{Id}\ A\ a = \mathsf{Id}\ B\ a}\ cong \qquad \dfrac{B = B \qquad y : a = b}{\mathsf{Id}\ B\ a = \mathsf{Id}\ B\ b}\ cong}{\mathsf{Id}\ A\ a = \mathsf{Id}\ B\ b}\ trans$$

Corbineau [10] notes this as an open problem. Of course, the above proof is unnecessarily complicated. The same equation can be proved by a single use of congruence. Furthermore, the simpler

$$
\begin{array}{lcl}
p & \mapsto & p \\
\mathsf{refl}^{-1} & \mapsto & \mathsf{refl} \\
\mathsf{refl}; p & \mapsto & p \\
p; \mathsf{refl} & \mapsto & p \\
(p; q); r & \mapsto & p; (q; r) \\
p; p^{-1} & \mapsto & \mathsf{refl} \\
p^{-1}; p & \mapsto & \mathsf{refl} \\
 & & \\
{p^{-1}}^{-1} & \mapsto & p \\
(p; q)^{-1} & \mapsto & q^{-1}; p^{-1}
\end{array}
\qquad
\begin{array}{rcl}
(\mathsf{cong}_{\ A}\ p_1 \mathinner{..} p_i)^{-1} & \mapsto & \mathsf{cong}_{\ A}\ {p_1}^{-1} \mathinner{..} {p_i}^{-1} \\
(\mathsf{inj}_i\ p)^{-1} & \mapsto & \mathsf{inj}_i\ (p^{-1}) \\
(\mathsf{cong}_{\ A}\ p_1 \dots p_i); (\mathsf{cong}_{\ A}\ q_1 \dots q_i) & \mapsto & \mathsf{cong}_{\ A}\ (p_1; q_1) \mathinner{..} (p_i; q_i) \\
\mathsf{inj}_k\ (\mathsf{cong}_{\ A}\ p_1 \dots p_i) & \mapsto & p_k \\
\mathsf{inj}_k\ ((\mathsf{cong}_{\ A}\ p_1 \mathinner{..} p_i); r) & \mapsto & p_k; (\mathsf{inj}_k\ r) \\
\mathsf{inj}_k\ (r; (\mathsf{cong}_{\ A}\ p_1 \dots p_i)) & \mapsto & (\mathsf{inj}_k\ r); p_k \\
x_{\triangleright(r; \mathsf{cong}_{\ =}\ p\ q)} & \mapsto & p^{-1}; (x_{\triangleright r}); q
\end{array}
$$

$$
\frac{p \mapsto p'}{x_{\triangleright p} \mapsto x_{\triangleright p'}}\textsc{Assumption}
\qquad
\frac{p \mapsto p' \quad q \mapsto q'}{p; q \mapsto p'; q'}\textsc{Trans}
\qquad
\frac{\forall k.\ \ p_k \mapsto p'_k}{\mathsf{cong}_{\ A}\ p_1 \mathinner{..} p_i \mapsto \mathsf{cong}_{\ A}\ p'_1 \mathinner{..} p'_i}\textsc{Cong}
\qquad
\frac{p \mapsto p'}{\mathsf{inj}_k\ p \mapsto \mathsf{inj}_k\ p'}\textsc{Inj}
$$

Figure 10. Simplification rules for evidence terms

proof does not have any issues with typing: every expression occurring in the derivation is either a subexpression of the goal or a subexpression of one of the equations from the context, so we know they are well-typed.

Our key observation is that this trick works in general. The only time a congruence proof will involve expressions which were not already present in the context or goal is when transitivity is applied to two derivations ending in cong. We simplify such situations using the following CONGTRANS rule.

$$(\mathsf{cong}_{\ A}\ p_1 \mathinner{..} p_i); (\mathsf{cong}_{\ A}\ q_1 \mathinner{..} q_i) \mapsto \mathsf{cong}_{\ A}\ (p_1; q_1) \mathinner{..} (p_i; q_i)$$

This rule is valid in general, and it does not make the proof larger. We also need rules for simplifying evidence terms that combine transitivity with injectivity or assumption-up-to-CC, such as $\mathsf{inj}_i\ \ (\mathsf{cong}_{\ A}\ p_1 \mathinner{..} p_k)$ and $x_{\triangleright(r; \mathsf{cong}_{\ =}\ p\ q)}$, rules for pushing uses of symmetry ($^{-1}$) past the other evidence constructors, and rules for rewriting subterms. The complete simplification relation $\mapsto$ is shown in Figure 10.

Any evidence term p can be simplified into a normalized evidence term p^*. (In the extended version of the paper we define an explicit grammar for fully simplified terms p^*, and prove than any term can be simplified into that form). And given p^* it is easy to produce a corresponding proof term in the core language. The idea is that one can reconstruct the middle expression in every use of transitivity $(p; q)$, because at least one of p and q will be specific enough to pin down exactly what equation it is proving. Formally, we define the judgement $\Gamma \vdash^{\mathsf{L}} p : a = b$ by adding evidence terms to the rules in Figure 9, and then prove:

***Lemma* 5.** If we have $\mathsf{label}\,\Gamma \vdash^{\mathsf{L}} p^* : \mathsf{label}\,a = \mathsf{label}\,b$ and $\Gamma \vdash a = b : \mathsf{Type}$, then $\Gamma \vDash a = b$.

Simplifying the evidence terms also solves another issue, which arises because of the TCCERASURE rule. Because the input terms are preprocessed to delete annotations (Section 7.1), an arbitrary evidence term will not uniquely specify the annotations. Again, this issue only arises because of the cong-trans pair. Simplifying the evidence term resolves the issue, because in a simplified term every intermediate expression is pinned down.

Putting together the labelling step, the evidence simplification step and the proof term generation step we can relate typed and untyped congruence closure. In the following theorem, the relation $\Gamma \vdash a = b$ is defined by similar rules as Figure 6 except that we omit the typing premises in TCCREFL, TTCERASURE and TTCCONGRUENCE.

***Theorem* 6.** Suppose $\Gamma \vdash a = b$ and $\Gamma \vdash a = b : \mathsf{Type}$. Then $\Gamma \vDash a = b$. Furthermore $\Gamma \vdash v : a = b$ for some v.

The computational content of the proof is how the elaborator generates core language evidence for equalities, so this shows the correctness of the ZOMBIE implementation. But it is also interesting as a theoretical result in its own right, and an important part of the proof of completeness of elaboration (Section 6).

8. Extensions

The full ZOMBIE implementation includes more features than the surface language described in Section 5. We omitted them from the formal system in order to simplify the proofs, but they are important to make programming up to congruence work well.

8.1 Smart case

Although we do not include datatypes in this paper, they are a part of the ZOMBIE implementation, and an important component of any dependently-typed language. The presence of congruence closure elaboration means that the core language [26] can use a specification of dependently-typed pattern matching called *smart case* [1].

With smart case, the rule for case analysis introduces a new equation into the context when checking each branch of a case expression. For example, the rule for an if expression type checks each branch under the assumption that the condition is true or false.

$$\frac{\begin{array}{c}\Gamma \vdash a : \mathsf{Bool} \\ \Gamma, x : a = \mathsf{true} \vdash b_1 : A \\ \Gamma, x : a = \mathsf{false} \vdash b_2 : A\end{array}}{\Gamma \vdash \mathsf{if}\ a\ \mathsf{then}\ b_1\ \mathsf{else}\ b_2 : A}\textsc{TFullCase}$$

This rule is in contrast to specifications that use *unification* to communicate the information gained by pattern matching. In those systems, if the scrutinee and the patterns are not unifiable (in the fragment of higher-order unification supported by the type system) then the case expression must be rejected. Furthermore, the specification of the typing rule for the unification based systems is more complicated. Smart case, by pushing this information to propositional equality, is both simpler and more expressive.

The downside to smart case has been that because this information is recorded as an assumption in the context, it is more work for the programmer. However, with congruence closure, the type system is immediately able to take advantage of these equalities in each branch. Thus, the ZOMBIE surface language has the convenience of the unification-based rule, while the core language enjoys the simplicity of smart case.

8.2 Reduction modulo congruence

In the paper all β-reductions are introduced by expressions `join : a = b`. But in practice some additional support from the typechecker for common patterns can make programming much more pleasant.

First, one often wants to evaluate some expression a "as far as it goes". Then making the programmer write both sides of the equation $a = b$ is unnecessarily verbose. Instead we provide the syntax `unfold a in body`. The implementation reduces a to normal form, $a \rightsquigarrow_{\mathsf{cbv}} a' \rightsquigarrow_{\mathsf{cbv}} a'' \rightsquigarrow_{\mathsf{cbv}} a'''$ (if a does not terminate the programmer can specify a maximum number of steps), and then introduces the corresponding equations into the context with fresh names. That is, it elaborates as

```
let _ = (join : a = a') in
let _ = (join : a' = a'') in
let _ = (join : a'' = a''') in
  body
```

Second, many proofs requires an interleaving of evaluation and equations from the context, particularly in order to take advantage of equations introduced by smart case. One example is `npluszero` in Section 2. The case-expression needs to return a proof of `n+0 = n`. If we try to directly evaluate `n+0`, we would reach the stuck expression `case n of Zero → 0; Succ m' → Succ (m' + 0)`, so instead we used an explicit type annotation in the `Zero` branch to evaluate `0+0`. However, the context contains the equation `n = Zero`, which suggests that there should be another way to make progress.

To take advantage of such equations, we add some extra intelligence to the way unfold handles CBV-evaluation contexts, that is expressions of the form $f\ a$ or $(\mathsf{case}\ b\ \mathsf{of} \ldots)$. When encountering such an expression it will first recursively unfold the function f, the argument a, or the scrutinee b (as with ordinary CBV-evaluation), and add the resulting equations to the context. However, it will then examine the congruence equivalence class of these expressions to see if they contain any suitable values—any value v is suitable for a, a function value $\mathsf{rec}\ f\ x.a_0$ for f, and a value headed by a data constructor for b—and then unfold the resulting expression $(\mathsf{rec}\ f\ x.a_0)\ v$. (If there are several suitable values, one is selected arbitrarily). This way unfolding can make progress where ordinary CBV-evaluation gets stuck.

Using the same machinery we also provide a "smarter" version of `join`, which first unfolds both sides of the equation, and then checks that the resulting expressions are CC-equivalent. This lets us omit the type annotations from `npluszero`:

```
npluszero n = case n [eq] of
                 Zero → smartjoin
                 Suc m → ...
```

The unfold algorithm does not fully respect CC-equivalence, because it only converts *into* values. For example, suppose the context contains the equation $f\ a = v$, Then $\mathsf{unfold}\ g\ (f\ a)$ will evaluate $f\ a$ and add the corresponding equations to the context, but $\mathsf{unfold}\ g\ v$ will not cause $f\ a$ to be evaluated. This gives the programmer more control over what expressions are run.

We have not studied the theory of the unfold algorithm, and indeed it is not a complete decision procedure for our propositional equality. If a subexpression of a does not terminate, unfold will spend all its reduction budget on just that subexpression (but this is OK, because the programmer decides what expression a to unfold). And if the context contains e.g. an equation between two unrelated function values, unfold will arbitrarily choose one of them (but it is hard to think of an example where this would happen). We have found unfold very helpful when writing examples.

9. Related work

The annotated core language in this paper is a slight variation on previous work [26], which in turn is a subset of the full language implemented by ZOMBIE [9]. In this version, in order to keep the formalism small we omit some features (uncatchable exceptions and general datatypes) and replace the application rule with a slightly less expressive value-dependent version. However, these omissions are not significant (the original system is still compatible with the "up to congruence" approach and is implemented in ZOMBIE). We also took the opportunity to simplify some typing rules, and to emphasize the role of erasable annotations.

Propositional Equality The idea of using congruence closure is not limited to the particular version of propositional equality used by our core language, which has some nonstandard features (we discussed the motivations for them in [26]). Below, we discuss how those features interact with congruence closure and suggest how the algorithm could be adapted to other settings.

First, our equality is *very heterogeneous*, that is we can form and use equations between terms of different types. This has pros and cons: it can be convenient for the programmer to not worry about types, and the metatheory is simple, but it makes it hard to include type-directed η-rules. However, congruence closure will work just as well with a conventional homogeneous equality.

Second, we use an *n-ary congruence rule*, while most theories only allow eliminating one equation at a time. For congruence closure to work equality must be a congruence, e.g. given $a = a'$ and $b = b'$ we should be able to conclude $f\ a\ b = f\ a'\ b'$. Our n-ary rule supports this in the most straightforward way possible. An alternative (used in some versions of ETT [11]) would be to use separate n-ary congruence rules for each syntactic form. Systems that only allow rewriting by one equation at a time require some tricks to avoid ill-typed intermediate terms (e.g. [6] Section 8.2.7).

Finally, in our system the elimination of propositional equality is *erased*, so equations like $a_{\triangleright b} = a$ are considered trivially true. This is similar to Extensional Type Theory, but unlike Coq and Agda. Having such equations available is important, because the elaborator inserts casts automatically, without detailed control by the programmer. In Coq that would be problematic, because an inserted cast could prevent two terms from being equal. However, making the conversion erasable is not the only possible approach. For example, in Observational Type Theory [2] the conversions are computationally relevant but the theory includes $a_{\triangleright b} = a$ as an axiom. In that system one can imagine the elaborator would use the axiom to make the elaborated program type-check.

Stronger equational theories The theory of congruence closure is one among a number of related theories. One can strengthen it in various ways by adding more reasoning rules, in order to get a more expressive programming language. However, doing so may endanger type inference, or even the decidability of type checking.

One obvious question is whether we could extend the relation $\Gamma \vDash a = b$ to do both congruence reasoning and β-reduction at the same time. Unfortunately, this extension causes the relation to become undecidable.

This is clearly the case in our language, which directly includes general recursive function definitions. But even if we allowed only terminating functions, the combination of equality assumption and lambdas can be used to encode general recursion. For example, reasoning in a context containing

```
f : Nat → Nat
h : f = (\x. if (even x) then f (n/2) else f (3*n+1))
```

is equivalent to having available a direct recursive definition

```
f x = if (even x) then f (n/2) else f (3*n+1)
```

Another natural generalization is to allow *rewriting by axiom schemes*, i.e. instead of only using ground equations $a = b$ from the context, also instantiate and use quantified formulas like $\forall xyz.a = b$. In general this generalization (the "word problem") is also not decidable, e.g. it is easy to write down an axiom scheme for the equational theory of SKI-combinators. However, there are semi-decision procedures such as *unfailing completion* [4] which form the basis of many automated theorem provers.

Even when preserving decidability one can still extend congruence closure to know about specific axioms schemes, such as for natural numbers with successor and predecessor [20] or lists [19] or injective data constructors [10].

Clearly one could design a programming language around a more ambitious theory than just congruence closure. Many languages, such as Dafny [17] and Dminor [7] call out to an off-the-shelf theorem prover in order to take advantage of all the theories that the prover implements. One reason we focus on a simple theory is that it makes *unification* easier, which seems to offer promising avenues for future work on type inference. Unification modulo congruence closure (rigid E-unification) is NP-complete [15]. This compares favorably with unification modulo β (higher-order unification) which is undecidable. Unification modulo other equational theories (E-unification) must be handled on a theory-by-theory basis, and it is not an operation exposed by most provers.

Simplifying congruence proofs Our CONGTRANS simplification rule is quite natural, and in fact the same rule has been studied before for a different reason. For efficiency, users of congruence closure want to make proofs as small as possible by taking advantage of simplifications like $\mathsf{refl}; p \mapsto p$ or $p^{-1}; p \mapsto \mathsf{refl}$ [13, 29]. However, uses of cong can hide the opportunity for such simplifications. De Moura et al. define the same CONGTRANS rule and give the following example [13]. Given assumptions $h_1 : a = b, h_2 : b = d, h_3 : c = b$, consider the proof term

$$(\mathsf{cong}_f\,(h_1; {h_3}^{-1})); (\mathsf{cong}_f\,(h_3; h_2)) : fa = fd$$

We can get rid of the assumption h_3 by doing the rewrite

$$(\mathsf{cong}_f\,(h_1; {h_3}^{-1})); (\mathsf{cong}_f\,(h_3; h_2)) \mapsto \mathsf{cong}_f\,(h_1; {h_3}^{-1}; h_3; h_2).$$

Dependent programming with congruence closure CoqMT [28] aims to make Coq's definitional equality stronger by including additional equational theories, such as Presburger arithmetic, so that for example the types $\mathsf{Vec}\ 0$ and $\mathsf{Vec}\ (n \times 0)$ can be used interchangeably. The prototype implementation only looks at the types themselves, but the metatheory also considers using assumptions from the context. This is complicated because CoqMT still wants to consider types modulo β-convertibility, and in contexts with inconsistent assumptions like $\mathsf{true} = \mathsf{false}$ one could write nonterminating expressions. Therefore CoqMT imposes restrictions on where an assumption can be used. VeriML makes the definitional equality user-programmable [27], and as an example builds a "stack" combining congruence closure, β-reduction, and potentially other theorem proving.

Neither CoqMT or VeriML prove that their implementation is complete with respect to a declarative specification. For example, the VeriML application rule requires that the applied function has the type $T \to T'$ and then checks that T is definitionally equal to the type of the argument, but there is no attempt to also handle declarative derivations which require definitional equality to create an arrow type.

The Guru language includes a tactic `hypjoin` [21] similar to our `smartjoin` and `unfold`. However, instead of using equations from the context, the programmer has to write an explicit list of equations, and unlike `unfold` it normalizes the given equations.

10. Conclusion

We consider this paper as an application of automatic theorem proving to language design. Of course, in a higher-order logic, we always expect that the programmer will have to supply *some* proofs manually; the question is which ones. Intensional Type Theory recognizes that $\beta\eta$-equivalence in a normalizing language is decidable, so such equality proofs can be handled automatically as part of the definitional equality relation. This paper considers a different decidable equational theory, and proposes a language that is "the dual of ITT": while conventional dependently-typed languages automatically use equalities that follow from β-reductions but do not automatically use assumptions from the context, our language uses assumptions but does not automatically reduce expressions.

We look forward to exploring the ramifications of this design decision more deeply in the context of a full programming language. Our ZOMBIE implementation provides a good baseline, but we would like to add more automation. In particular, the addition of rigid E-unification seems promising. Furthermore, we would like to explore ways in which β-reduction and congruence closure can co-exist—perhaps there is some way to achieve the benefits of each approach in the same context.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant Nos. 0910500, 1116620, and 1319880. The ZOMBIE implementation was developed with the assistance of the Trellys team. This paper was written with the help of the Ott tool [23]. The authors would also like to thank the anonymous reviewers for their comments.

References

[1] T. Altenkirch. The case of the smart case: How to implement conditional convertibility? Presentation at NII Shonan seminar 007, Japan, Sept. 2011.

[2] T. Altenkirch, C. McBride, and W. Swierstra. Observational equality, now! In *PLPV '07: Programming Languages meets Program Verification*, pages 57–68. ACM, 2007.

[3] L. Augustsson. Cayenne – a language with dependent types. In *ICFP '98: International Conference on Functional Programming*, pages 239–250. ACM, 1998.

[4] L. Bachmair, N. Dershowitz, and D. A. Plaisted. Completion Without Failure. In A. H. Kaci and M. Nivat, editors, *Resolution of Equations*

in Algebraic Structures, volume 2: Rewriting Techniques, pages 1–30. Academic Press, 1989.

[5] B. Barras and B. Bernardo. The Implicit Calculus of Constructions as a Programming Language with Dependent Types. In *11th international conference on Foundations of Software Science and Computational Structures (FOSSACS 2008)*, volume 4962 of *LNCS*, pages 365–379. Springer, 2008.

[6] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development, Coq'Art:the Calculus of Inductive Constructions*. Springer-Verlag, 2004.

[7] G. M. Bierman, A. D. Gordon, C. Hritcu, and D. E. Langworthy. Semantic subtyping with an SMT solver. In *ICFP '10: International Conference on Functional Programming*, pages 105–116, 2010.

[8] E. C. Brady. Idris—systems programming meets full dependent types. In *PLPV'11: Programming languages meets program verification*, pages 43–54. ACM, 2011. ISBN 978-1-4503-0487-0.

[9] C. Casinghino, V. Sjöberg, and S. Weirich. Combining proofs and programs in a dependently typed langauge. In *POPL '14: 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2014.

[10] P. Corbineau. Deciding equality in the constructor theory. In T. Altenkirch and C. McBride, editors, *Types for Proofs and Programs*, volume 4502 of *Lecture Notes in Computer Science*, pages 78–92. Springer Berlin Heidelberg, 2007.

[11] K. Crary. *Type-Theoretic Methodology for Practical Programming Languages*. PhD thesis, Cornell University, 1998.

[12] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[13] L. de Moura, H. Rueß, and N. Shankar. Justifying equality. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 125(3):69–85, July 2005.

[14] P. J. Downey, R. Sethi, and R. E. Tarjan. Variations on the common subexpression problem. *J. ACM*, 27(4):758–771, Oct. 1980.

[15] J. Gallier, W. Snyder, P. Narendran, and D. Plaisted. Rigid E-unification is NP-complete. In *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS '88)*, pages 218–227, 1988.

[16] L. Jia, J. A. Vaughan, K. Mazurak, J. Zhao, L. Zarko, J. Schorr, and S. Zdancewic. AURA: A programming language for authorization and audit. In *ICFP '08: International Conference on Functional Programming)*, pages 27–38, 2008.

[17] K. R. M. Leino. Dafny: an automatic program verifier for functional correctness. In *Proceedings of the 16th international conference on Logic for programming, artificial intelligence, and reasoning*, LPAR'10, pages 348–370. Springer-Verlag, 2010.

[18] C. McBride. First-order unification by structural recursion, 2001.

[19] G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2):356–364, Apr. 1980.

[20] R. Nieuwenhuis and A. Oliveras. Fast congruence closure and extensions. *Inf. Comput.*, 205(4):557–580, Apr. 2007.

[21] A. Petcher and A. Stump. Deciding Joinability Modulo Ground Equations in Operational Type Theory. In S. Lengrand and D. Miller, editors, *Proof Search in Type Theories (PSTT)*, 2009.

[22] B. C. Pierce and D. N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, Jan. 2000.

[23] P. Sewell, F. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strnisa. Ott: Effective tool support for the working semanticist. *J. Funct. Program.*, 20(1):71–122, 2010.

[24] R. E. Shostak. An algorithm for reasoning about equality. *Commun. ACM*, 21(7):583–585, July 1978.

[25] V. Sjöberg and S. Weirich. Programming up to congruence (extended version). Technical Report MS-CIS-14-10, University of Pennsylvania, Philadelphia, PA, Oct. 2014.

[26] V. Sjöberg, C. Casinghino, K. Y. Ahn, N. Collins, H. D. Eades III, P. Fu, G. Kimmell, T. Sheard, A. Stump, and S. Weirich. Irrelevance, heterogeneous equality, and call-by-value dependent type systems. In J. Chapman and P. B. Levy, editors, *MSFP '12*, volume 76 of *EPTCS*, pages 112–162. Open Publishing Association, 2012.

[27] A. Stampoulis and Z. Shao. Static and user-extensible proof checking. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 273–284, New York, NY, USA, 2012. ACM.

[28] P.-Y. Strub. Coq modulo theory. In *CSL*, pages 529–543, 2010.

[29] A. Stump and L.-Y. Tan. The algebra of equality proofs. In *16th International Conference on Rewriting Techniques and Applications (RTA'05)*, pages 469–483. Springer, 2005.

[30] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang. Secure Distributed Programming with Value-dependent Types. In *ICFP '11: International Conference on Functional Programming*, pages 285–296. ACM, 2011.